Tutorial 2

Week of January 22, 2007

1 Scheme on CDF

```
Invoking:                 scheme

Exiting:                  (exit)   or    Ctrl-D

Loading filename.scm:     (load ‘‘filename’’)
                           or
                          (load ‘‘filename.scm’’)

Tracing:                  (trace proc_name)

Transcript:

  (transcript-on <my_transcript>)
  (transcript-off)

will save a transcript of a session to <my_transcript>.

Debugger:
   -start:                              (debug)
   -help:                               ?
   -go back (read-eval-print level):    (restart 1)  or Ctrl-C Ctrl-C
   -quit:                               q
```

2 Read-Eval-Print loop

1. Read input from user
2. Evaluate input
3. Print return value

HOW DOES EVALUATION GO? Let's see...

```
1 ]=> 1
;Value: 1

this means:  ``1'' evaluates to 1

1 ]=> +
;Value 1: #[arity-dispatched-procedure 1]

This means ``+'' evaluates to some addition procedure. STRESS THIS.
```

```
1 ]=> (+ 1 2)

;----------------------------------------------------------------------
; Attention!!! I see an opening bracket!!!
;
; Can I find a matching closing bracket? Yes. OK. Moving along.
; (If I don't find a matching closing bracket, I'll just complain
;    and do nothing)
;
; Here's what I do next:
;
;   - I look at the first thing after an opening bracket and I
;     evaluate it. I NEED it to evaluate to a procedure. If it does
;     not, I complain and do nothing.
;     Here I am looking at the string ``+''. I can evaluate it to a
;     procedure which does the addition (as above). OK. Moving along.
;
;   - Now I look at the rest of the things before the closing bracket
;     and evaluate them. If I cannot evaluate something along the way,
;     I complain and do nothing. Otherwise, move along.
;     Well, in our case ``1'' evaluates to 1 and ``2'' evaluates to 2.
;     So far, so good.
;
;   - Now I attempt to apply the addition procedure (which I got from
;     evaluating ``+'') to 1 and 2, which I got from evaluating ``1''
;     and ``2''. It happens so that the addition procedure is defined
;     in such a way that it successfully does the job of adding 1 and
;     2 and it says that 3 is the answer. I am done. I have evaluated
;     the expression ``(+ 1 2)'' to 3. So I say so.
;----------------------------------------------------------------------
;Value: 3
```

3 Defining Procedures

Not all procedures are pre-defined (like addition in the previous example). Here's how we define procedures. In what follows, we show how to define a procedure, *without* associating a name with it. There are many interesting uses for such "unnamed procedures" which we will see later in the course. Later in this tutorial we will see how to associate a name with such a procedure definition.

```
(lambda (var1 var2 ... varN) exp1 exp2 ... expM)

``lambda'' is a {\em syntactic form}, a special keyword that tells the
interpreter that what follows is a procedure definition.

var1, var2, ..., varN are the arguments to the procedure.
The rest is the body of the procedure.
All of exp1, exp2, ..., expM are evaluated; value of expM only is returned.

1 ]=> (lambda (x) x)
;Value 1: #[compound-procedure 1]

``(lambda (x) x)'' got evaluated to a procedure that takes something
as a argument and returns it.

1 ]=> ( lambda (x) (+ x x) )
;Value 1: #[compound-procedure 2]

''( lambda (x) (+ x x) )'' got evaluated to a procedure that takes
something as an argument and returns the result of evaluating
``(+ something something)''.
```

```
1 ]=>( (lambda (x) (+ x x)) 5 )
;Value: 10

``( (lambda (x) (+ x x)) 5 )'' is evaluated as follows:

  - see an opening bracket

  - evaluate the first thing after it:
        ``(lambda (x) (+ x x))'' evaluates to a procedure as above

  - ``5'' evaluates to 5

  - the procedure is applied to 5:
         - evaluate ``(+ 5 5)'' :
               - evaluate ``+'' to an addition procedure
               - evaluate 5 to 5
               - get 10

        - return the result of evaluating ``(+ 5 5)'', i.e. return 10

  - print the result: ``;Value: 10''

BUT NOTE:

1 ]=>  ( (lambda (x y) (* x y) (+ x y)) 3 5)
; Value: 8
```

4 Giving a name to a procedure

```
(define proc-name (lambda (var1 var2 ... varN) exp1 exp2 ... expM))
```

"define", like "lambda" is a *syntactic form.*

```
When I see ``define'', I know I will
have 2 things following it, before the closing bracket.

Then the following happens:
 - the second thing is evaluated
 - the first thing gets the value obtained from evaluation

1 ]=> (define my_proc (lambda (x) (+ x x)))
;Value: my_proc

This means:
  - ``(lambda (x) (+ x x))'' got evaluated to a procedure as before
  - my_proc now has a value: it is that procedure

1 ]=> (my_proc 5)
;Value: 10

this means:  - ``my_proc'' got evaluated to a procedure above
             - apply that procedure to 5
             - get 10 and print it
```

IMPORTANT NOTE:

This is no different from:

```
1 ]=> (define x 10)
;Value: x

1 ]=> x
;Value: 10
```

We evaluate ‘‘10’’ to 10 and assign it to x. In case of procedure definition the second thing just happens to evaluate to a procedure.

Shortcut:

```
  (define (proc-name var1 ... varN) exp1 ... expM)
```

It is THE SAME AS

```
  (define proc-name (lambda (var1 var2 ... varN) exp1 exp2 ... expM))
```

```
1 ]=> (define (my_proc x) (+ x x))
;Value: my_proc

1 ]=> (my_proc 5)
;Value: 10
```

5 Example

We saw this example in lecture. In the file called "myfile.scm" you have the following:

```
----------------------------
; my comments are here
(define (increment n)
        (+ n 1))

; some other comments are here
(define foobar 21)
----------------------------

werewolf:~\% scheme
Scheme Microcode Version 14.9
MIT Scheme running under GNU/Linux
Type `^C' (control-C) followed by `H' to obtain information about interrupts.
Scheme saved on Monday June 17, 2002 at 10:03:44 PM
  Release 7.7.1
  Microcode 14.9
  Runtime 15.1

1 ]=> (load "myfile")

;Loading "myfile.scm" -- done
;Value: foobar

1 ]=> (increment foobar)

;Value: 22

OK. What happened there?
1) Loading
2) Evaluation
```

```
1) Loading:
   same as typing in the stuff in the interpreter:
   we have defined the procedure ``increment'' and
   we have defined foobar to be 21

2) Evaluating ``(increment foobar)'':

 - lookup procedure value corresponding to ``increment'':
   (lambda (n) (+ n 1))

 - lookup ``foobar'': 21

 - evaluate ``( (lambda (n) (+ n 1)) 21 )'':
    - evaluate (+ 21 1)
        - evaluate ``+'' to an addition procedure
        - evaluate 21 to 21 and 1 to 1
        - apply addition procedure to 21 and 1, get 22
 - print 22


And Scheme interpreter sort of tells you this:

1 ]=> (trace increment)
;Unspecified return value    <---- just means ``nothing is returned''
                                   it is not an error, we can procede

1 ]=> (increment foobar)

[Entering #[compound-procedure 1 increment]
    Args: 21]
[22
      <== #[compound-procedure 1 increment]
    Args: 21]
;Value: 22

Well, you have to learn its language...
```

6 Quoting

`quote` is another *syntactic form*. It inhibits the normal evaluation of the expression that follows it.

```
(quote <expression>)  is same as '<expression>

'<expression>  evaluates to <expression>

Some examples:

1 ]=> (define x 1)
;Value: x

1 ]=> 'x
;Value: x

1 ]=> x
;Value: 1

1 ]=> '1
;Value: 1

1 ]=> 1
;Value: 1


More interesting:

(define x 'y)
(define y  5)
(define z  y)

(+ y 1)    evals to   6
(+ z 1)    evals to   6
(+ x 1)    evals to   Error

EXPLAIN WHAT HAPPENS HERE IN DETAIL
```